# Implementación de MCTS y Redes Convolucionales para el desarrollo de una red nueronal en R

Anthonny Flores,....
Escuela de Matemáticas
Universidad de Costa Rica
San José, Costa Rica
Email: anflores@ucr.ac.cr

Abstracto

## I. Introduccion.

## II. Metodología.

Para el bot de ajedrez Hatchet1, el equipo se basó en el Aprendizaje por Refuerzo Profundo (Deep Reinforcement Learning), el cual toma inspiración del modelo AlphaZero para lograr un autoentrenamiento continuo (Self-Play). Este enfoque reemplaza el conocimiento heurístico humano programado con una Red Neuronal Profunda (NN) que se mejora iterativamente, guiando un potente algoritmo de Búsqueda de Árbol Monte Carlo (MCTS).

### 1. Red Neuronal Profunda y Codificación del Estado (NN)

El diseño de la Red Neuronal y la precisión de la Codificación del Estado son los pilares de la inteligencia de Hatchet1.

#### 1.1. Codificación del Estado (FEN a Tensor) - Función fen.to.vector

La función fen.to.vector es esencial para la interfaz entre el motor de ajedrez (chess en R) y la NN. Transforma el estado FEN en un tensor $8 \times 8 \times 18$ (64 casillas, 18 canales), que es el formato de entrada esperado por keras3.

1.1.1. Canales de Piezas (1-12): Se utiliza una codificación one-hot para las 12 piezas. La asignación se realiza por la tabla map.piece, donde las piezas blancas se mapean a los canales 1 a 6 ("P"=1 a "K"=6) y las negras a los canales 7 a 12 ("p"=7 a "k"=12). El bucle recorre el FEN (rows) y asigna el valor 1 al canal correspondiente a la pieza y casilla.

1.1.2. Canales de Metadatos (13-18): Estos canales proporcionan información global que no se infiere de la distribución de piezas, siendo idénticos en todas las casillas ($8 \times 8$).

- Canal 13 (Turno): Valor 1 si es el turno de las blancas ("w"), 0 si es el turno de las negras.
- Canal 14 (Enroque): Valor 1 si el enroque es legal para al menos un lado.
- Canal 15 (Repetición): Valor 1 si se detectan dos repeticiones de posición, anticipando la regla de tres repeticiones.
- Canal 16 (Constante): Una capa constante de 1, utilizada para ayudar a la red a detectar el límite del tablero.
- Canal 17 (Reloj de Medio Movimiento): El contador de movimientos sin captura o avance de peón (board.data[5]). Este valor se normaliza entre 0 y 1, utilizando un divisor de 100.0 ($\min(1, \text{clock}/100.0)$), crucial para que la NN aprenda la proximidad de la Regla de 50 Movimientos.
- Canal 18 (Número de Movimiento Completo): El contador total de movimientos (board.data[6]). Se normaliza con un divisor de 200.0, proporcionando una medida de la profundidad de la partida.

Finalmente, el array se reajusta (array_reshape) para coincidir con el tensor de lote requerido por keras3 para la predicción del modelo.

#### 1.2. Arquitectura, Profundidad y Funciones de Activación

El cuerpo de la NN está compuesto por una Capa Convolucional Inicial y 20 Bloques Residuales (num.residual.blocks). La profundidad del cuerpo, con 20 bloques y 256 filtros, es crucial para la fuerza

del modelo, similar a la arquitectura Hatchet1 de la competencia.

La elección de 20 Bloques Residuales permite a la red aprender representaciones jerárquicas complejas del tablero. Un Bloque Residual es fundamental en esta arquitectura para mitigar el problema del vanishing gradient al permitir que la información "salte" capas y facilitar el flujo del gradiente durante el entrenamiento.

Funciones de Activación ReLU: La función de activación principal utilizada en todas las capas internas del cuerpo y dentro de los bloques residuales es la Unidad Lineal Rectificada (ReLU). Esta función introduce no-linealidad de manera eficiente, ya que es simple y ayuda a mantener el entrenamiento estable.

$$\text{ReLU}(x) = \max(0, x)$$

Cabezales de Salida (Output Heads): Al final del cuerpo principal, la red se bifurca en dos cabezales de salida independientes, cada uno utilizando una función de activación final diferente:

- Cabezal de Política (Policy Head, $p$): Predice la probabilidad de cada movimiento legal. Utiliza la función Softmax para asegurar que la suma de las probabilidades de todas las jugadas legales sea 1.
- Cabezal de Valor (Value Head, $V$): Predice el resultado esperado de la posición para el jugador de turno, un valor entre $-1$ (derrota) y $+1$ (victoria). Utiliza la función Tangente Hiperbólica (Tanh):

$$\text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

  La función Tanh es seleccionada debido a que su rango de salida $[-1, 1]$ coincide directamente con la escala de los resultados del ajedrez, facilitando la cuantificación de la calidad posicional.

### 1.3. Optimización y Ponderación de Pérdida

La compilación del modelo utiliza el optimizador Adam con una tasa de aprendizaje de 0.0001. La Función de Pérdida es una combinación de mean_squared_error (para el valor $V$) y categorical_crossentropy (para la política $p$). Ambas pérdidas se ponderan con 1.0 (loss_weights = list(value = 1.0, policy = 1.0)), indicando que el modelo debe optimizar por igual la precisión de la calidad del movimiento y la predicción del resultado.

## 2. Búsqueda de Árbol Monte Carlo (MCTS)

El MCTS es el algoritmo de decisión central, implementado en la función run.mcts y dependiente de la función best.move. Para el equipo, el MCTS es el corazón del agente; es el que toma la recomendación de la NN y la mastica 20 veces (o las que se le haya configurado en NUM.SIMULATIONS) para entregar la mejor jugada.

### 2.1. Constantes de Control Global

Dos constantes globales controlan el equilibrio de la búsqueda:

- $C.PUCT = 2.0$: El Coeficiente de Exploración. Este valor, definido globalmente, es super importante porque es la balanza que decide si se va a explorar caminos nuevos (porque la NN dice que son prometedores, alta $P$) o si se va a explotar caminos que ya se conocen y que han dado buen resultado (alto valor $Q$). Es el factor de riesgo, por así decirlo.
- $NUM.SIMULATIONS = 20$: Número de simulaciones realizadas por MCTS por cada movimiento. Esto es un tira y encoge entre lo profundo que se desea pensar y el tiempo disponible para mover. Con 20 simulaciones, se le da una revisada decente a las opciones sin eternizarse.

El árbol de búsqueda se almacena en el entorno global mcts.tree, actuando como una hash map para acceder rápidamente a las estadísticas de nodo $(N, W, P)$ a través de la clave FEN. Esto es como tener un cuaderno de apuntes global donde se guarda cuánto se visita un estado $(N)$, cuánto se gana desde ahí $(W)$, y la probabilidad inicial $(P)$ que le dio la red.

### 2.2. Fases de MCTS y Mecanismos de Robustez

La función run.mcts encapsula las cuatro fases de la simulación, y es donde se hace el trabajo pesado del análisis.

2.2.1. Selección con PUCT: La selección se guía por la función calculate.ucb, que usa la fórmula PUCT. Esta fórmula es la que guía el recorrido en el árbol:

$$\text{UCB} = Q + C_{PUCT} \cdot P \cdot \frac{\sqrt{N_{\text{sum}}}}{1 + N_{\text{current}}}$$

El valor $Q$ es lo que se ha visto que funciona (la parte de Explotación), mientras que el segundo término, donde entra $C_{PUCT}$ y $P$, es la parte de Ex-

ploración. Se elige siempre el movimiento que dé el UCB más alto. El valor $Q$ es calculado como el ratio de valor/visitas ($W/N$) del nodo actual.

2.2.2. Expansión y Parche de Asignación Segura: Si se llega a un nodo nuevo, uno que no se ha visitado (o sea, $\sum N = 0$), se hace la Expansión. Se obtiene la predicción de la NN para ese nuevo estado. Aquí, es vital el Parche de Asignación Segura que se le configuró, porque la red a veces se vuelve loca o tiene un recorte de política que suma poco:

1. Se recorta la política $P_{\text{model}}$ solo a los movimientos legales.
2. Se verifica que la suma de esas probabilidades no sea insignificante (o sea, mayor que safety.threshold, que es $1e-4$).
3. Si es buena la suma, se Normaliza ($P \leftarrow P/\sum P$).
4. Si la suma es muy baja, no se confía en la red en este punto, y es mejor asignar una Política UNIFORME ($P \leftarrow 1/\text{num\_moves}$) a todos los movimientos. Esto, aunque parezca simple, es lo que evita que el algoritmo se quede pegado y garantiza que siga explorando robustamente.

2.2.3. Retropropagación y Anulación de Recompensa: Una vez que se llega al final de la simulación (ya sea por fin de partida o por la predicción de la NN en un nodo expandido), toca actualizar. La Retropropagación actualiza las estadísticas de visita y valor. El valor de recompensa ($Z$) se va pasando hacia atrás. Ojo: para el jugador anterior, el valor se invierte ($Z \leftarrow -Z$) porque lo que es bueno para un jugador es malo para su oponente, y viceversa. Esta inversión se hace en cada paso del bucle (for (node.data in rev(path))).

- Penalización de Empates: Se le puso un castigo sutil a los empates para que el agente no juegue tan aburrido. Si se llega a un empate por tablas (is_stalemate, is_seventyfive_moves, o is_fivefold_repetition), el valor se anula a $Z = -0.5$ en lugar de 0. Es una pequeña penalización para que prefiera la victoria.

### 2.3. Estrategia de Auto-Juego y Control de Muestreo - Función best.move

Esta función es la que toma el resultado del MCTS . También es crucial para el Self-Play (autoentrenamiento).

- Ruido de Dirichlet: Si la partida está en las primeras 30 jugadas (move.count $< 30$), se le inyecta Ruido de Dirichlet a la política en la raíz. Esto es como darle un empujón para que pruebe jugadas que la NN no ve como las mejores, promoviendo una exploración más amplia al inicio.
- Temperatura de Muestreo (high.temp): Esta es la famosa "temperatura" que controla la exploración vs. explotación a lo largo del juego:
  1. Modo Exploración (Jugadas iniciales, move.count $< 40$): Se usa Temperatura $T = 1.0$. Aquí, el movimiento no es solo el más visitado, sino que se elige aleatoriamente (sample) basado en las probabilidades de visita ($P_{\text{MCTS}}$). Así se asegura que se exploren buenas jugadas que no son necesariamente la mejor en ese momento.
  2. Modo Explotación (Juego avanzado, move.count $\geq 40$): Simplemente se toma la jugada con el máximo número de visitas. Es la hora de jugar a ganar.
- Controles de Seguridad: La robustez es clave. Si por alguna razón los valores de $P_{\text{MCTS}}$ salen raros (NA, 0, o si el which.max falla), el bot no se rompe; simplemente elige un movimiento completamente aleatorio (sample(moves.list, size = 1)).

## 3. Generación, Almacenamiento y Evaluación de Datos

El sistema de datos (datos_ajedrez.R) y las librerías (chess, processx) garantizan la experiencia de juego persistente y analizable. Este módulo es nuestra "memoria" y es vital para que el agente Hatchet1 pueda aprender a largo plazo, ya que aquí se guarda toda la experiencia de auto-juego para el próximo ciclo de entrenamiento.

### 3.1. Estructuras de Datos y Persistencia

Se utilizan las funciones df.game.data y df.heavy.game.data para registrar los datos. La primera se encarga de guardar lo esencial de la partida, mientras que la segunda, df.heavy.game.data, es la que almacena la información vital y más pesada.

La función data.to.txt se encarga de que la data no se pierda. Guarda los archivos de juego en un directorio específico de almacenamiento con el argumento append = TRUE. Esto es clave, porque cada partida nueva se agrega al final del archivo existente,

asegurando el crecimiento continuo del conjunto de entrenamiento sin sobrescribir nada.

- Datos Pesados (games.heavy.data): Aquí registramos información crítica. Por ejemplo:
  - Historial FEN (Tabla_totales): ¡Esto es oro! Es la secuencia de todas las posiciones de la partida, crucial para recrear y entrenar a la NN.
  - Tiempo de Movimiento Medio (Tiempo_movimiento_medio): No solo jugamos bien, sino que controlamos cuánto nos cuesta. Este tiempo se calcula usando la mediana de los tiempos de CPU dedicados al MCTS en cada jugada, dándonos una idea real de la carga computacional.
- Además, el ELO del agente y el color con el que jugó se registran también en esta estructura.

### 3.2. Medición del Rendimiento

El rendimiento se cuantifica mediante el sistema ELO, calculado por la función elo.function. Este es nuestro termómetro de fuerza. El agente se evalúa frente a un oponente base con un ELO de 400.

Para la actualización del ELO, se usa la fórmula tradicional, pero con un factor de peso $K = 20$ (nuestra constante de ajuste, que tan rápido queremos que cambie nuestro ELO con cada resultado).

$$R'_{\text{Hatchet1}} = R_{\text{Hatchet1}} + 20 \cdot (\text{Score} - E_{\text{Esperada}})$$

Aquí, $R'_{\text{Hatchet1}}$ es el nuevo ELO, $R_{\text{Hatchet1}}$ es el ELO actual, Score es el resultado real (que puede ser 1 por victoria, 0.5 por empate, o 0 por derrota), y $E_{\text{Esperada}}$ es la probabilidad de victoria esperada contra el oponente (que sale de una fórmula basada en la diferencia de ELO). La lógica del cálculo se asegura de que estos resultados se utilicen correctamente en la fórmula, dándonos un feedback numérico preciso sobre qué tan bien está jugando Hatchet1.

## III. Resultados

## IV. Discusión

## V. Conclusiones

## Referencias

[1]